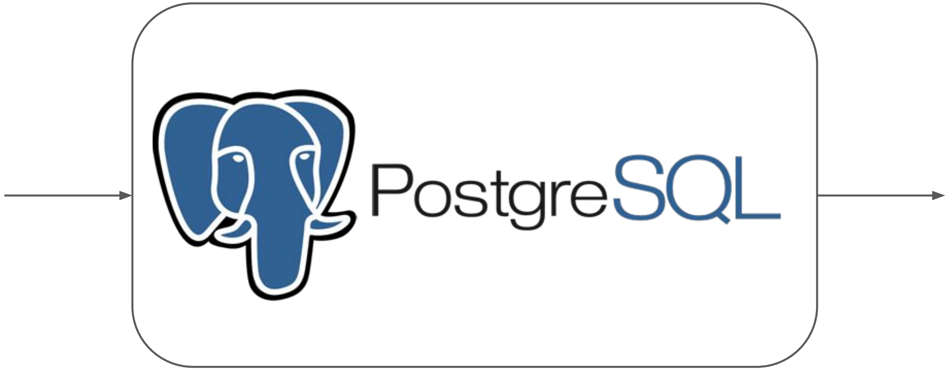
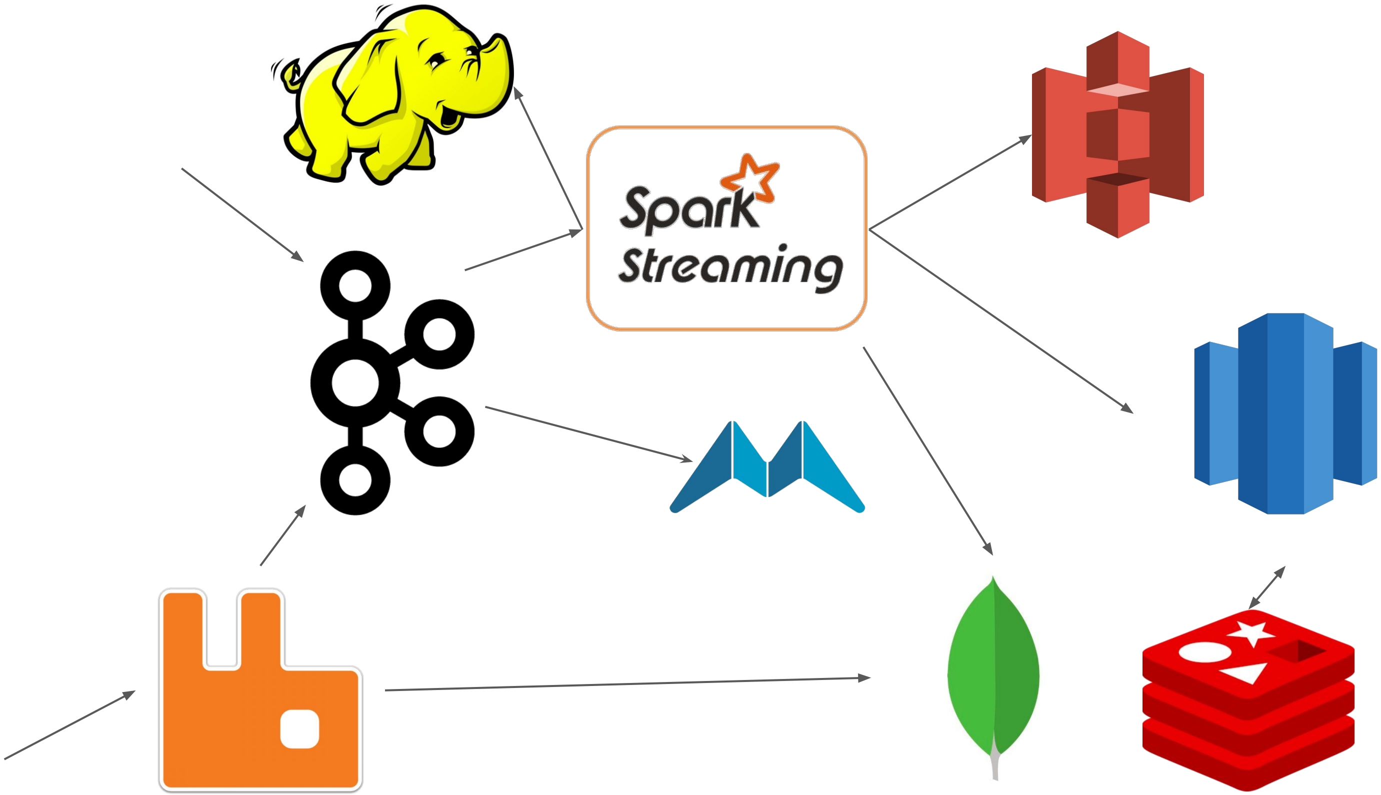
Distributed Computing on PostgreSQL

Marco Slot <marco@citusdata.com>

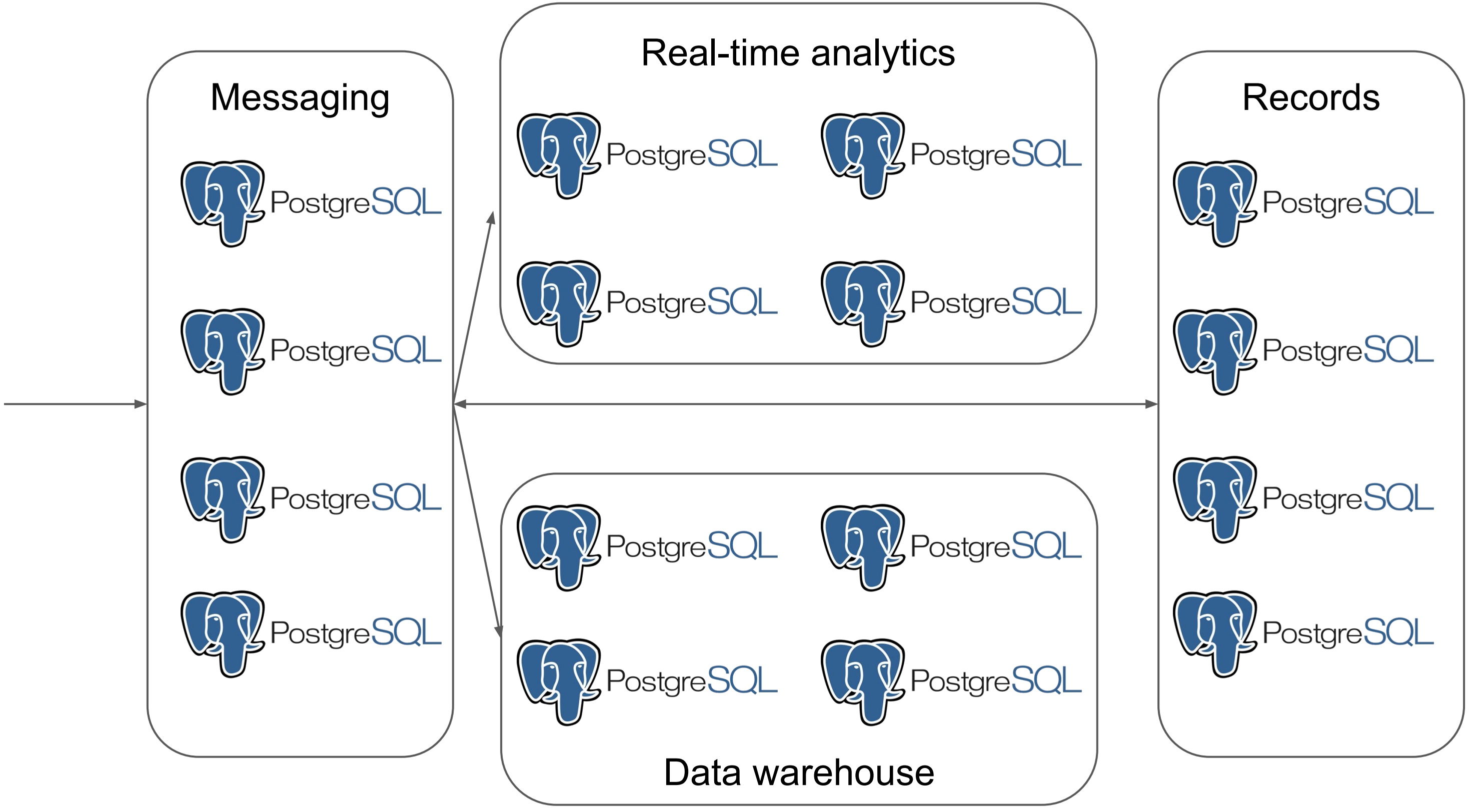
Small data architecture



Big data architecture



Big data architecture using postgres



PostgreSQL is a perfect building block
for distributed systems

Features!

PostgreSQL contains many useful features for building a distributed system:

- Well-defined protocol, libpq
- Crash safety
- Concurrent execution
- Transactions
- Access controls
- 2PC
- Replication
- Custom functions
- …

Extensions!

Built-in / contrib:

- postgres_fdw
- dblink **RPC!**
- plpgsql

Third-party open source:

- pglogical
- pg_cron
- citus

Extensions!

Built-in / contrib:

- postgres_fdw
- dblink **RPC!**
- plpgsql

Third-party open source:

- pglogical
- pg_cron
- citus

Yours!

dblink

Run queries on remote postgres server

```
SELECT dblink_connect(node_id,
    format('host=%s port=%s dbname=postgres', node_name, node_port))
FROM nodes;

SELECT dblink_send_query(node_id, $$SELECT pg_database_size('postgres')$$)
FROM nodes;

SELECT sum(size::bigint)
FROM nodes, dblink_get_result(nodes.node_id) AS r(size text);

SELECT dblink_disconnect(node_id)
FROM nodes;
```

RPC using dblink

For every postgres function, we can create a client-side stub using dblink.

```
CREATE FUNCTION func(input text)
  ...

CREATE FUNCTION remote_func(host text, port int, input text) RETURNS text
LANGUAGE sql AS $function$
  SELECT res FROM dblink(
    format('host=%s port=%s', host, port),
    format('SELECT * FROM func(%L)', input))
  AS res(output text);
$function$;
```

PL/pgSQL

Procedural language for Postgres:

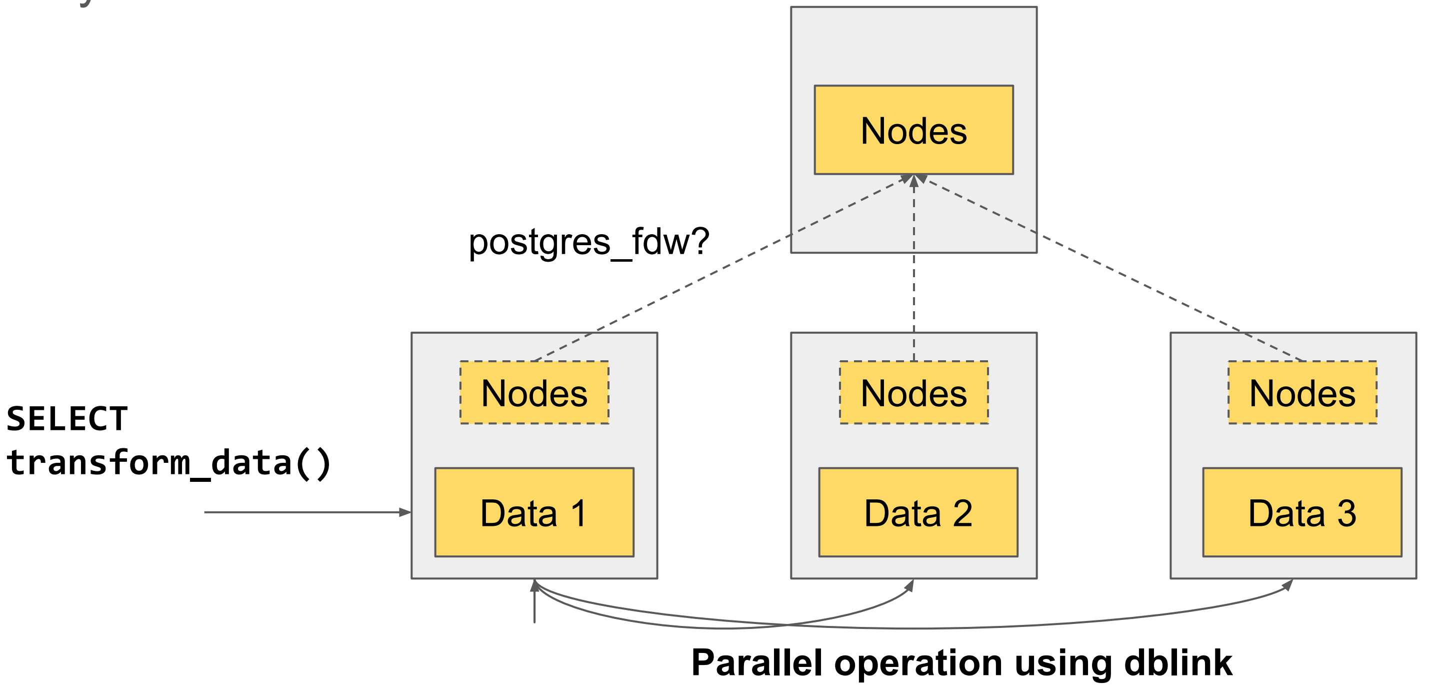
```
CREATE FUNCTION distributed_database_size(dbname text)
RETURNS bigint LANGUAGE plpgsql AS $function$
DECLARE
  total_size bigint;
BEGIN
  PERFORM dblink_send_query(node_id, format('SELECT pg_database_size(%L)', dbname)
  FROM nodes;

  SELECT sum(size::bigint) INTO total_size
  FROM nodes, dblink_get_result(nodes.node_id) AS r(size text);

  RETURN total_size
END;
$function$;
```

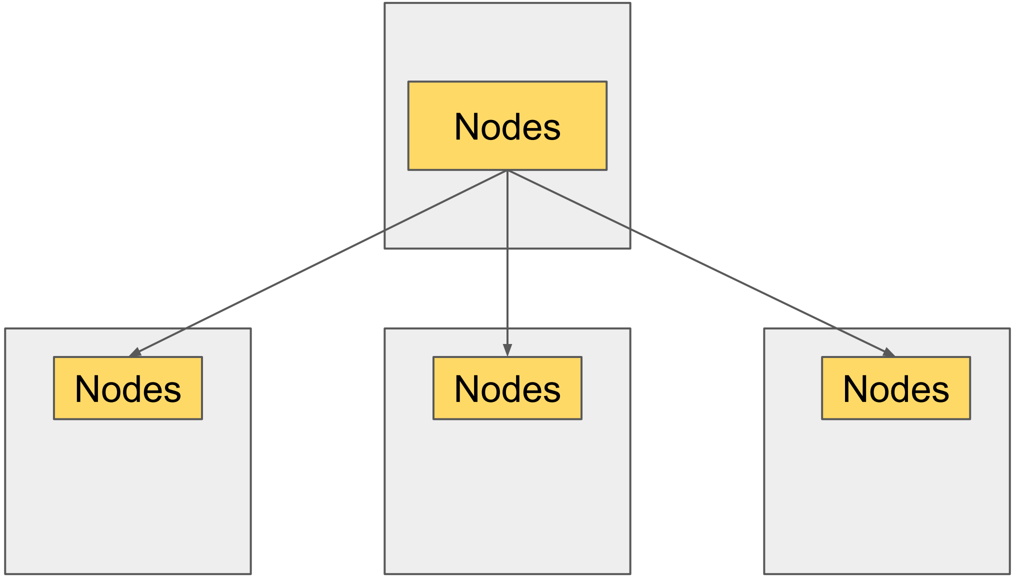
Distributed system in progress...

With these extensions, we can already create a simple distributed computing system.



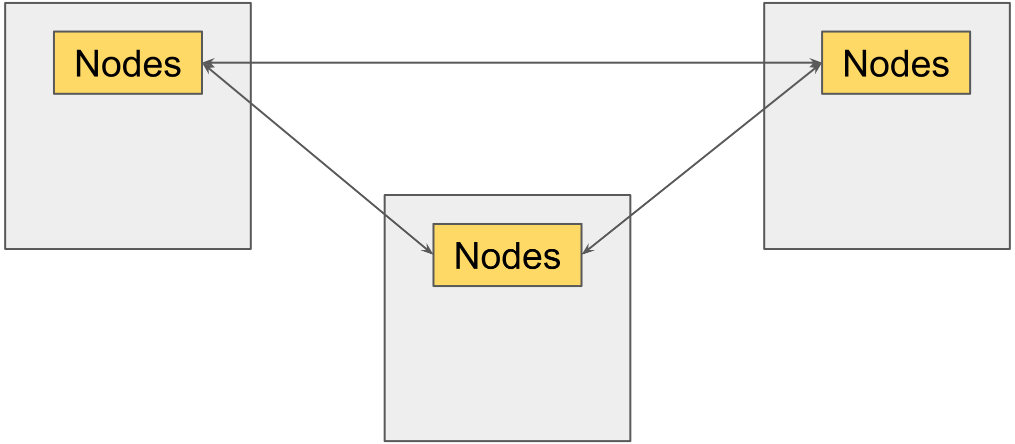
pglogical / logical replication

Asynchronously replicate changes to another database.



pg_paxos

Consistently replicate changes between databases.



pg_cron

Cron-based job scheduler for postgres:

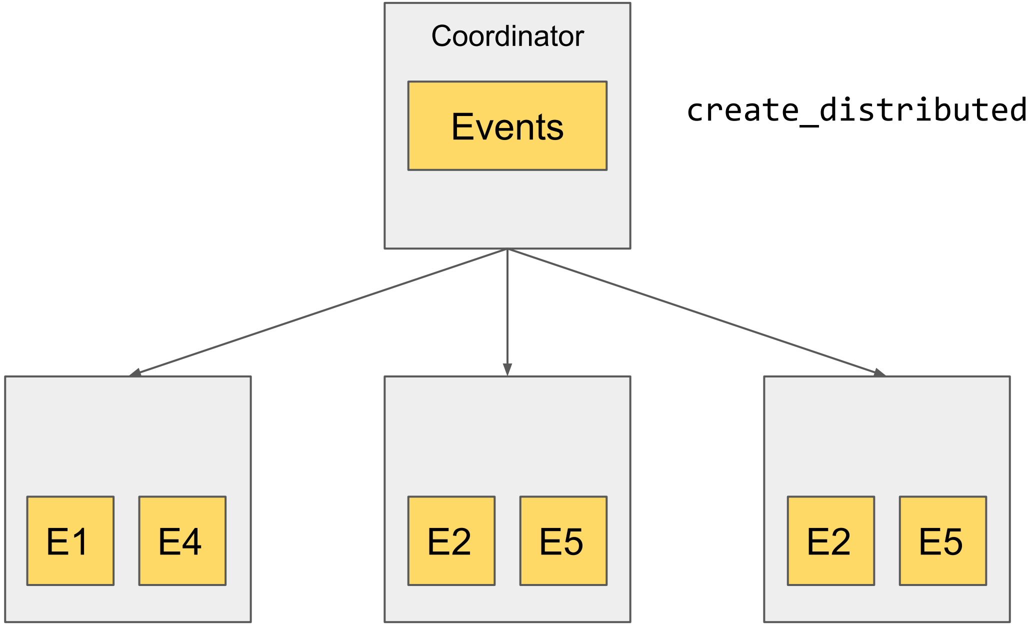
```
CREATE EXTENSION pg_cron;
SELECT cron.schedule('* * * * */10', 'SELECT transform_data()');
```

Internally uses libpq, meaning it can also schedule jobs on other nodes.

pg_cron provides a way for nodes to act autonomously

Citus

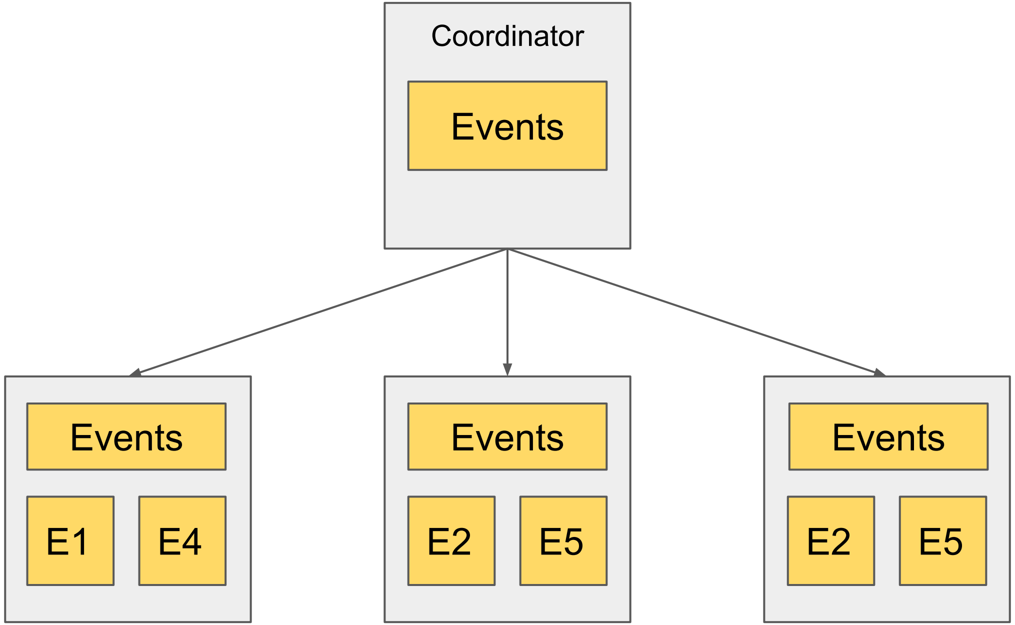
Transparently shards tables across multiple nodes



```
create_distributed_table('events',
                         'event_id');
```

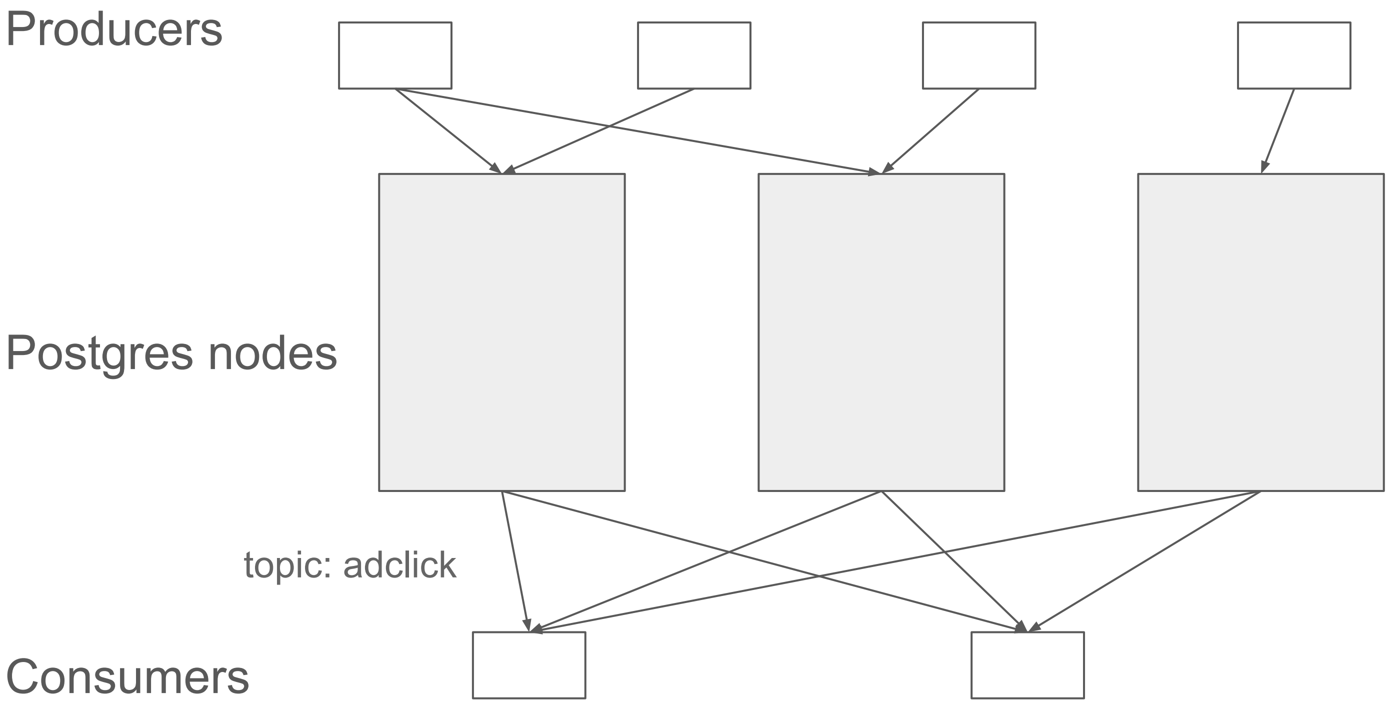
Citus MX

Nodes can have the distributed tables too

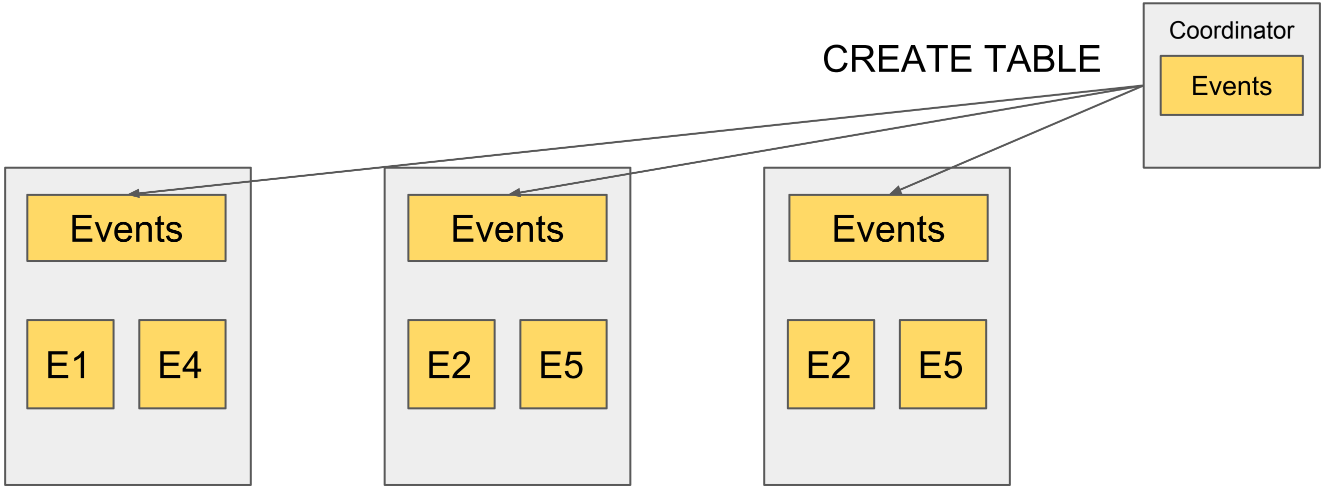


How to build a distributed system using only PostgreSQL & extensions?

Building a streaming publish-subscribe system



Storage nodes



Use Citus to create a distributed table

Distributed Table Creation

```
$ psql -h coordinator

CREATE TABLE events (
  event_id bigserial,
  ingest_time timestamptz default now(),
  topic_name text not null,
  payload jsonb
);
SELECT create_distributed_table('events', 'event_id');

$ psql -h any-node

INSERT INTO events (topic_name, payload) VALUES ('adclick','{...}');
```

Sharding strategy

Shard is chosen by hashing the value in the partition column.

Application-defined:

- **`stream_id`**`text not null`

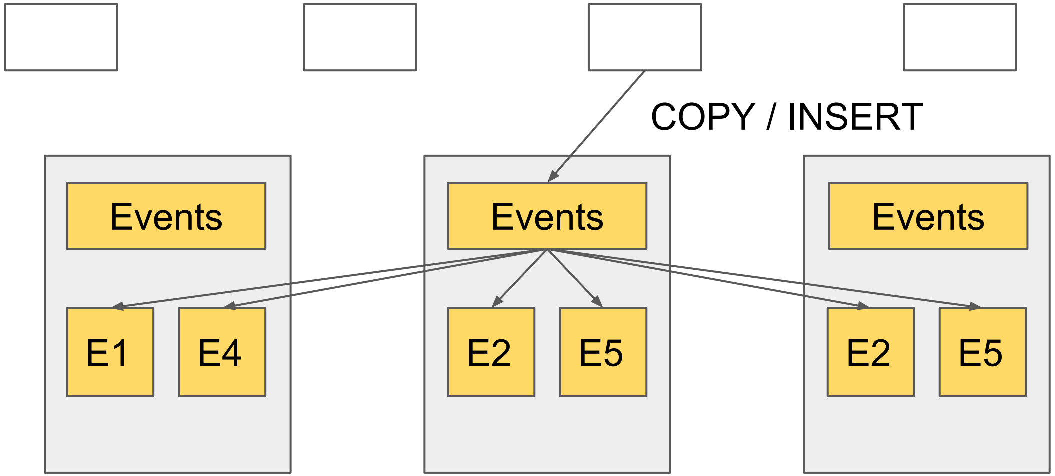
Optimise data distribution:

- **`event_id`**`bigserial`

Optimise ingest capacity and availability:

- **`sid`**`int default pick_local_value()`

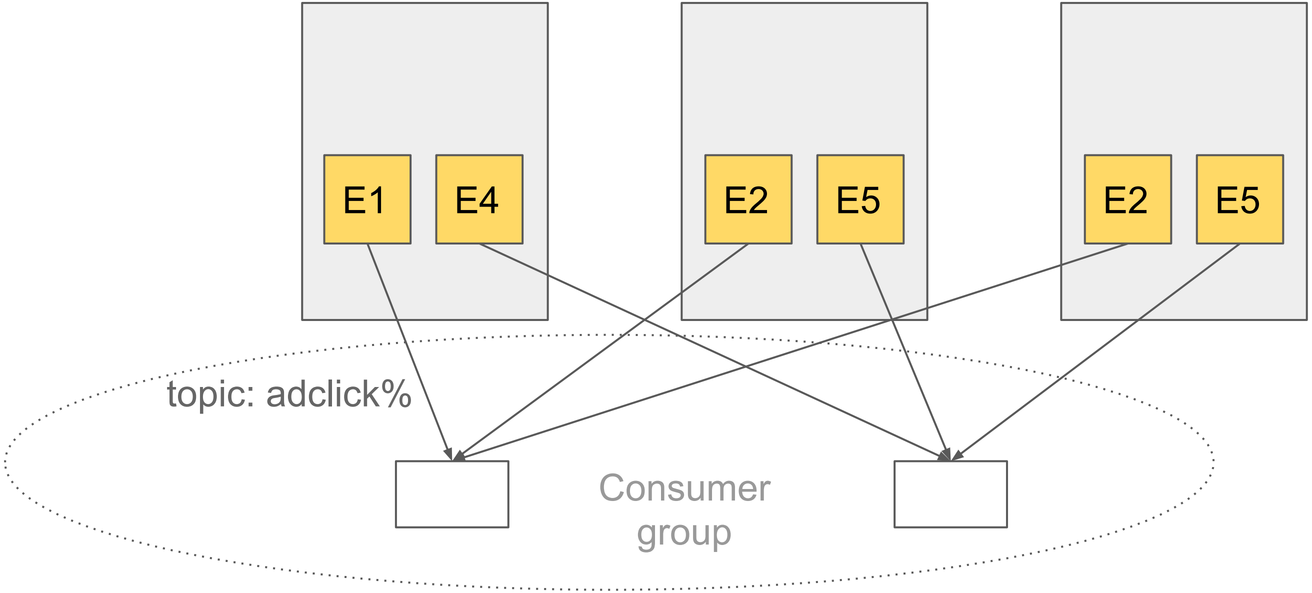
Producers



Producers connect to a random node and perform COPY or INSERT into events

Consumers

Consumers in a group together consume events at least / exactly once.



Consumer leases

Consumers obtain leases for consuming a shard.

Lease are kept in a separate table on each node:

```
CREATE TABLE leases (
  consumer_group text not null,
  shard_id bigint not null,
  owner text,
  new_owner text,
  last_heartbeat timestamptz,
  PRIMARY KEY (consumer_group, shard_id)
);
```

Consumer leases

Consumers obtain leases for consuming a shard.

```
SELECT * FROM claim_lease('click-analytics', 'node-2', 102008);
```

Under the covers: Insert a new lease or set **new_owner** to steal lease.

```
CREATE FUNCTION claim_lease(group_name text, node_name text, shard_id int)
  …
  INSERT INTO leases (consumer_group, shard_id, owner, last_heartbeat)
  VALUES (group_name, shard, node_name, now())
  ON CONFLICT (consumer_group, shard_id) DO UPDATE
  SET new_owner = node_name
  WHERE leases.new_owner IS NULL;
```

Distributing leases across consumers

Distributed algorithm for distributing leases across nodes

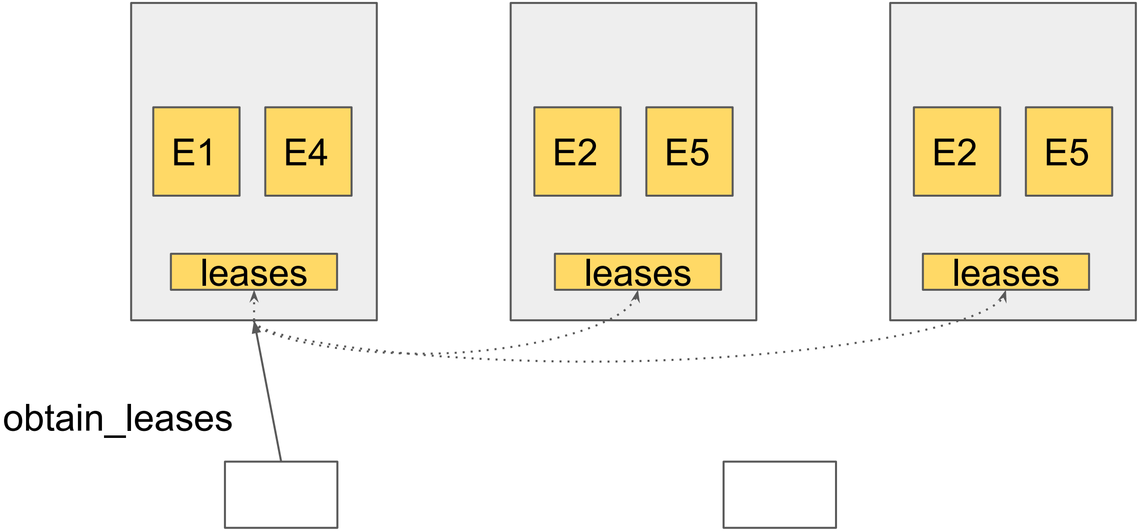
```
  SELECT * FROM obtain_leases('click-analytics', 'node-2')

   -- gets all available lease tables
   -- claim all unclaimed shards
   -- claim random shards until #claims >= #shards/#consumers
```

Not perfect, but ensures all shards are consumed with load balancing (unless C>S)

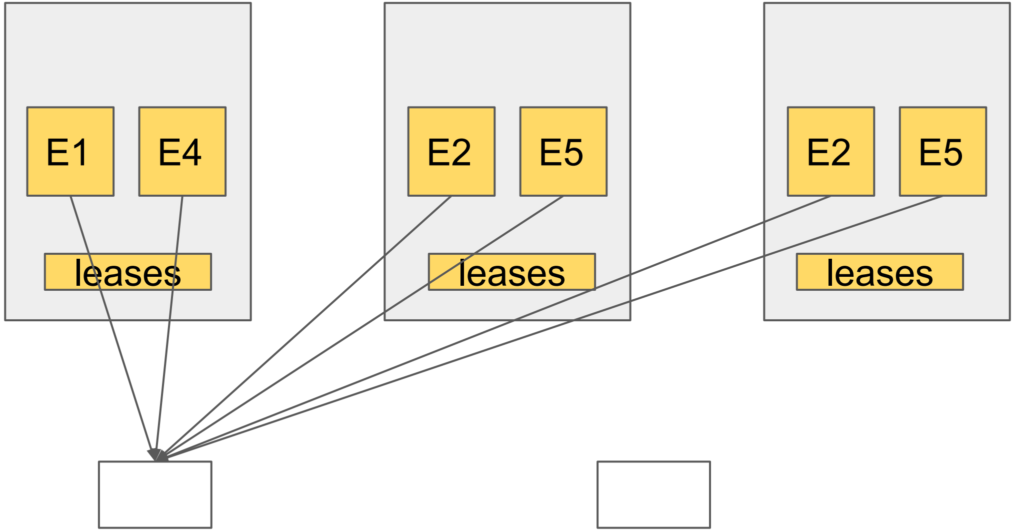
Consumers

First consumer consumes all



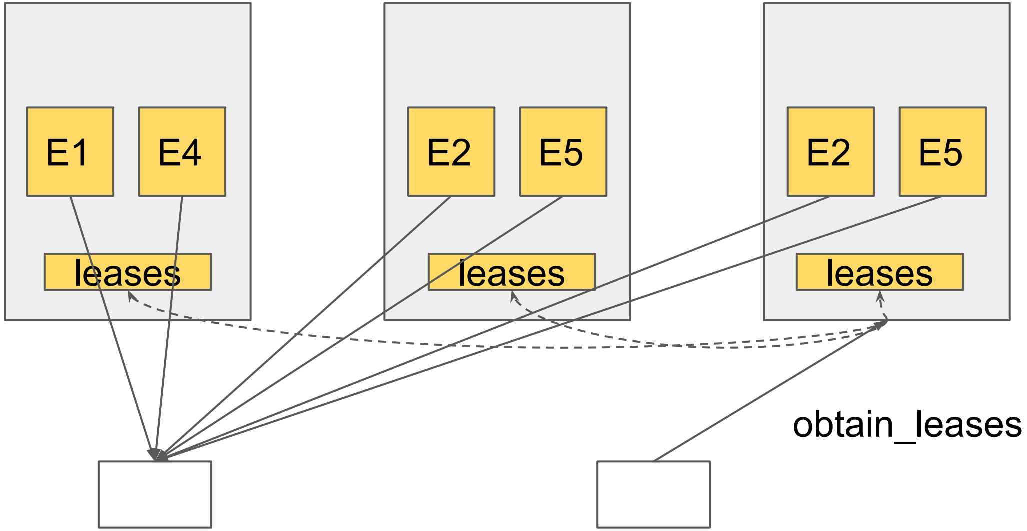
Consumers

First consumer consumes all



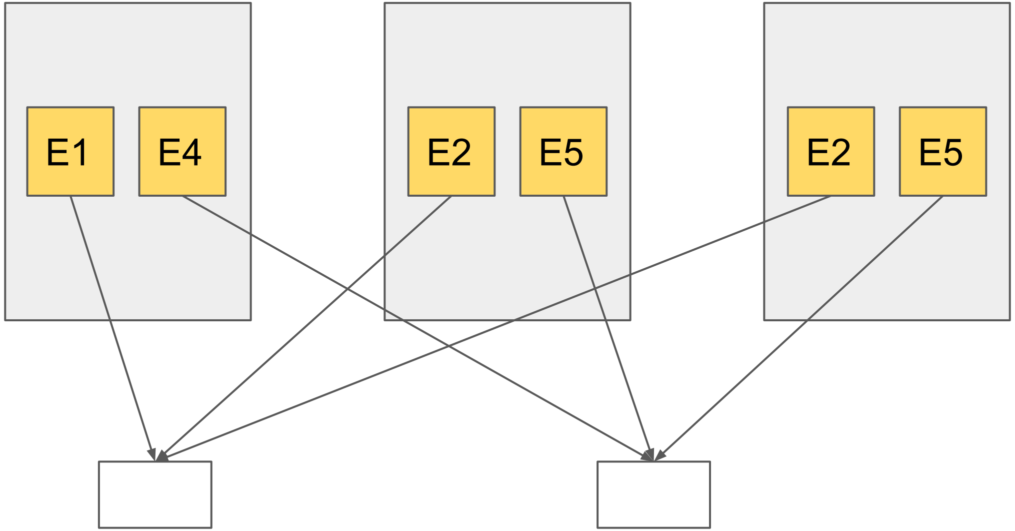
Consumers

Second consumer steals leases from first consumer



Consumers

Second consumer steals leases from first consumer



Consuming events

Consumer wants to receive all events once.

Several options:

- SQL level
- Logical decoding utility functions
- Use a replication connection
- PG10 logical replication / pglogical

Consuming events

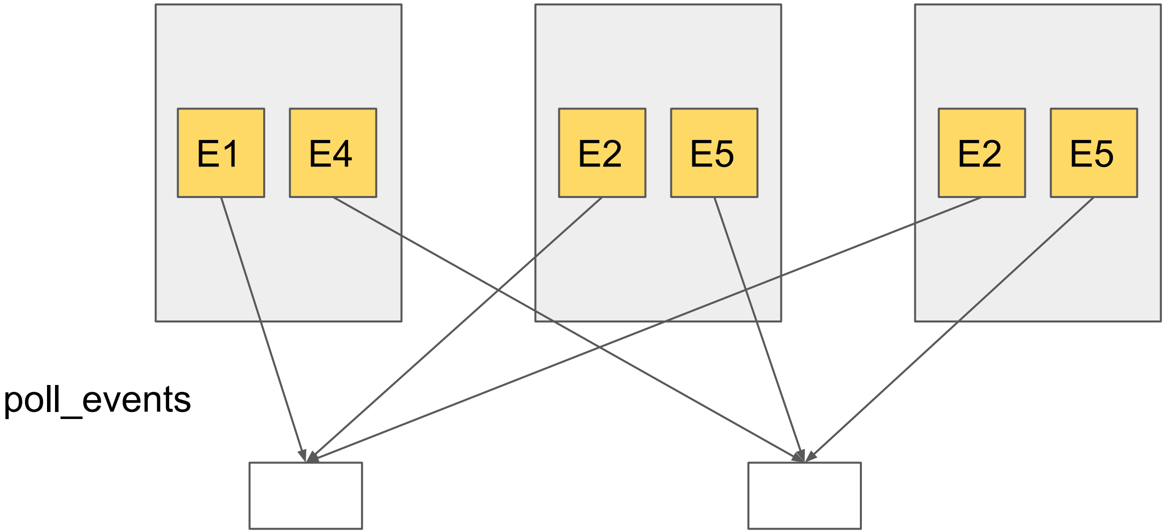
Get a batch of events from a shard:

```
SELECT * FROM poll_events('click-analytics', 'node-2', 102008, 'adclick',
                          '<last-processed-event-id>');
```

```
-- Check if node has the lease
   Set owner = new_owner if new_owner is set
-- Get all pending events            (pg_logical_slot_peek_changes)
-- Progress the replication slot  (pg_logical_slot_get_changes)
-- Return remaining events if still owner
```

Consumer loop

1. Call `poll_events` for each leased shard
2. Process events from each batch
3. Repeat with event IDs of last event in each batch



Failure handling

Producer / consumer fails to connect to storage node:
→ Connect to different node

Storage node fails:
→ Use `pick_local_value()` for partition column, failover to hot standby

Consumer fails to consume batch
→ Events are repeated until confirmed

Consumer fails and does not come back
→ Consumers periodically call `obtain_leases`
→ Old leases expire

Maintenance: Lease expiration

Use pg_cron to periodically expire leases on coordinator:

```
SELECT cron.schedule('* * * * *', 'SELECT expire_leases()');

CREATE FUNCTION expire_leases()
...
  UPDATE leases
  SET owner = new_owner, last_heartbeat = now()
  WHERE last_heartbeat < now() - interval '2 minutes'
```

Maintenance: Delete old events

Use pg_cron to periodically expire leases on coordinator:

```
$ psql -h coordinator

SELECT cron.schedule('* * * * *', 'SELECT expire_events()');

CREATE FUNCTION expire_events()
...
  DELETE FROM events
  WHERE ingest_time < now() - interval '1 day';
```

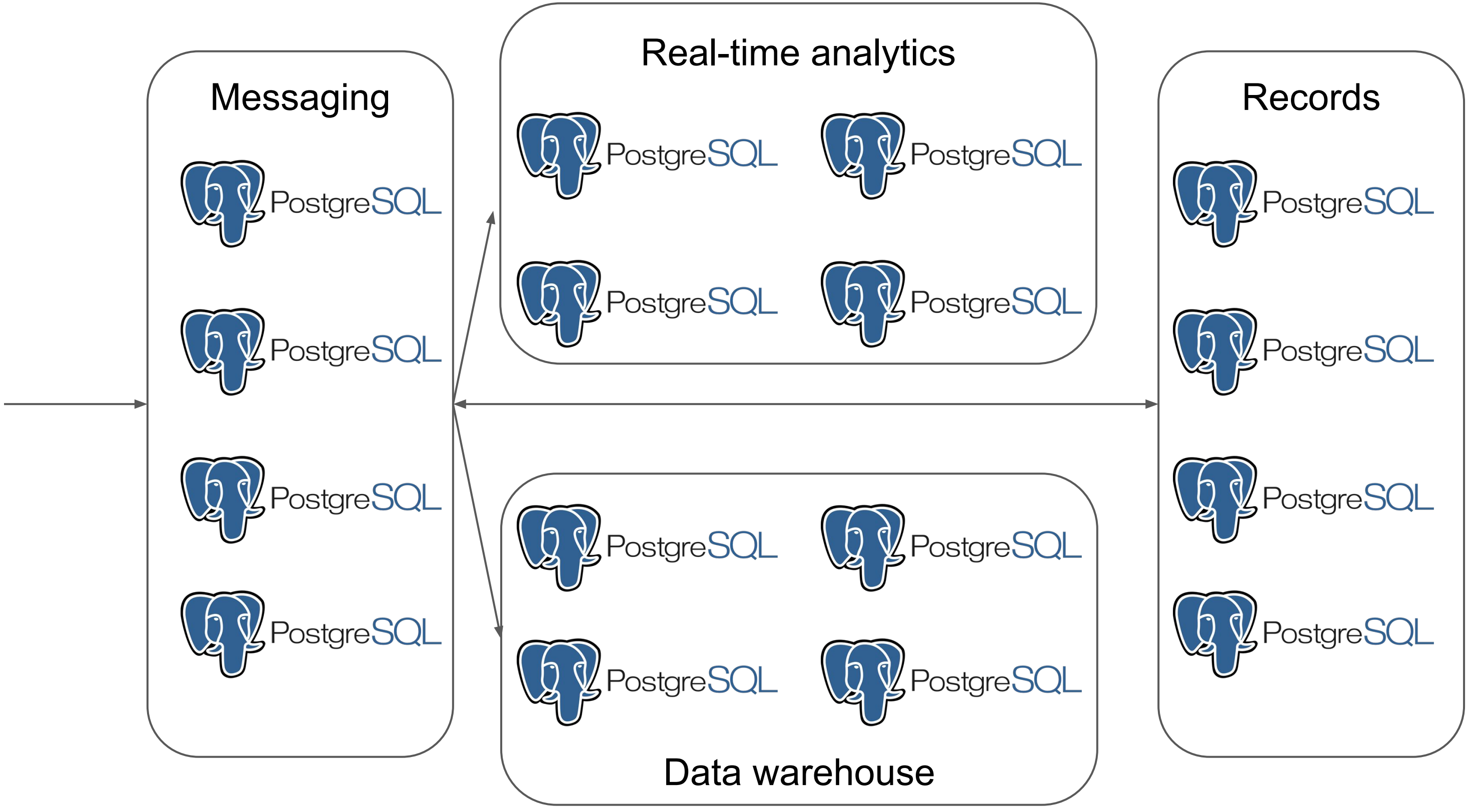
Prototyped a functional, highly available publish-subscribe systems in

~300 lines of code

https://goo.gl/R1suAo

Demo

Big data architecture using postgres



Questions?

marco@citusdata.com